

Project Computervisie: Plaats- / Schilderijherkenning

Arne Diaz, Emiel Vandenbussche,
Felix de Müelenaere & Youssef El Badaoui

UGent

Inhoudsopgave

Project Computervisie:

Plaats- / Schilderijherkenning	1
<i>Arne Diaz, Emiel Vandenbussche, Felix de Müelenrae & Youssef El Badaoui</i>	
1 Inleiding	3
2 Methode	3
2.1 Segmenteren van schilderijen	3
2.2 Matching Methode	6
2.2.1 ORB en Brute Force Matching	6
2.2.1.1 Oriented FAST	6
2.2.1.2 Rotated BRIEF	6
2.2.1.3 Brute Force Matching	7
2.2.2 Verkiezingsproces	7
2.2.3 Threading	7
2.2.3.1 In demo bestand	7
2.2.3.2 Op testset	8
2.2.4 Wazige afbeeldingen detecteren	8
2.2.4.1 Methode	9
2.3 Database oprichten	9
3 Resultaten	10
4 Conclusie	12
Bibliografie	13

1 Inleiding

Voor wie het *Museum voor Schone Kunsten* in Gent al is gaan bezoeken, kan het aanvoelen als een echt doolhof! Er zijn talloze zalen en als je kinderen hebt zouden ze in het museum verdwaald kunnen raken.

Daarom is dit project tot stand gekomen, de gebruiker kan door een foto te nemen van een willekeurig schilderij in een zaal te weten komen in welke zaal hij zich momenteel bevindt.

Bij zuivere foto's wordt de juiste zaal in 89% van de gevallen terug gegeven. Bij videobeelden in 70% van de gevallen.

Dit project legt volgende onderdelen grondig uit:

- detecteren en segmenteren van schilderijen
- een database opstellen met nuttige info over schilderijen
- een gelijkaardigheidswaarde van 2 schilderijen berekenen
- efficient de juiste zaal zoeken door te matchen met de database

2 Methode

2.1 Segmenteren van schilderijen

De eerste stap is het segmenteren van mogelijke schilderijen in de foto. Het segmenteren kan opgedeeld worden in drie delen: achtergrondkleur detectie, contouren detecteren en contouren uitknippen.

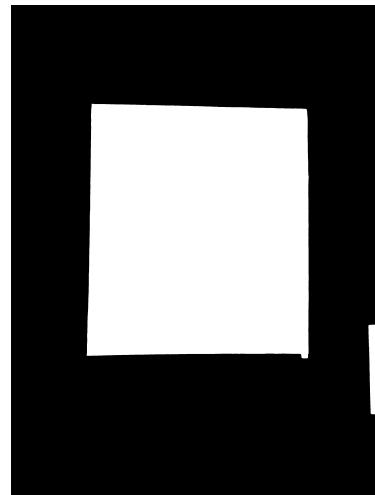
Eerst wordt de afbeelding 3x zo klein gemaakt als de resolutie groter is dan 1920 op 1080 pixels, dit om het floodFill algoritme te versnellen. Dan wordt de afbeelding geblurred met een Gaussian blur met een kernelgrootte van 21 op 21 pixels vooraleer de achtergrondkleur bepaald wordt. Om de achtergrondkleur te detecteren wordt de methode floodFill gebruikt. FloodFill vraagt 7 argumenten: de afbeelding, een lege matrix die als mask dient, een startpunt, een vervangkleur, 2 thresholds en een flag. De methode zoekt oppervlaktes die ongeveer dezelfde kleur hebben en geeft hier een nieuwe kleur aan. Het bepalen of 2 kleuren dezelfde zijn gebeurt aan de hand van 2 thresholds, een low threshold die bepaalt hoeveel lager de te vergelijken kleur mag zijn en een high threshold die bepaalt hoeveel hoger de te vergelijken kleur mag zijn. De methode start bij een pixel en vergeleikt de naburige pixels. Het aantal naburige pixels dat vergeleken wordt, wordt bepaald door de ingestelde vlag. In onze toepassing is de vlag op 4 ingesteld, waardoor 4 naburige pixels worden vergeleken (boven, onder, links en rechts). De vlag bepaalt ook hoe de mask-matrix ingevuld wordt. De mask-matrix dient om te weten welke pixels reeds toegekend zijn aan een oppervlak en dus niet meer geprocessed moeten worden. FloodFill geeft meerdere return values, maar voor deze toepassing is enkel de grootte van het ingevulde oppervlak van belang.

Een kopie van de originele afbeelding wordt pixel per pixel overlopen in een

lus en floodFill wordt toegepast op de huidige pixel als deze nog niet in een vorige iteratie verwerkt is. Een random kleur wordt gegenereerd om de pixels in te kleuren die samen een oppervlak vormen. Als floodFill een pixel inkleurt wordt de bijhorende waarde van deze pixel in de mask op 1 gezet, hierdoor weet het programma dat die pixel al verwerkt is. De kleur van het grootste oppervlak wordt bijgehouden. Aan het einde van de lus wordt een mask gegenereerd op de gekleurde afbeelding met de opgeslagen kleur. De nieuwe mask is zwart waar de achtergrondkleur zich bevond, de rest van het oppervlak is wit zoals te zien in figuur 1.



(a) originele afbeelding.



(b) mask.

Figuur 1: vergelijking origineel en gegenereerde mask

In de tweede stap worden de contouren gedetecteerd in de gegenereerde mask. Dit gebeurt met de methode findContours. Deze methode verwacht 3 argumenten. De afbeelding waarop de contouren bepaald moeten worden, een retrieval mode om te kiezen of je hiërarchie informatie terug wil krijgen en een contour approximation method waarmee je kan kiezen hoe de gevonden contouren voorgesteld worden. In deze toepassing is gekozen voor approximation method CHAIN_APPROX_NONE, dit betekent dat alle punten van een gevonden contour teruggegeven worden. Elk gevonden contour wordt getest op de grootte van zijn oppervlakte, als deze groter is dan een bepaalde waarde wordt ook nog gekeken hoe dicht deze contour zich bij de rand bevindt. Als de punten zich voor meer dan 40% aan de rand van de afbeelding bevinden, wordt dit contour niet gebruikt. Dit om te vermijden dat vloeren ook uitgesneden worden in de volgende stap. De contouren die overblijven zijn mogelijke schilderijen.

In de laatste stap worden de mogelijke schilderijen uitgesneden. Dit door eerst de omliggende rechthoek te bepalen van alle punten in een contour. Daarna wordt het mogelijk schilderij uitgeknippt en een affine transformatie gedaan om de proporties te corrigeren. Elk mogelijk schilderij wordt toegevoegd aan een lijst. Deze lijst wordt doorgegeven aan het matching algoritme. Het resultaat van het segmenteren van de originele afbeelding in figuur1 is te zien in figuur2.



Figuur 2: resultaat segmentatie op originele afbeelding.

2.2 Matching Methode

2.2.1 ORB en Brute Force Matching

Het ORB algoritme [2] is een combinatie van de FAST keypoint detector en de BRIEF descriptor die ook een betere performantie heeft dan BRIEF door een paar optimalisaties.

Het algoritme bepaalt keypoints en een beschrijving van hun omgeving (descriptoren).

2.2.1.1 Oriented FAST

Het FAST (Features from Accelerated Segment Test) algoritme is een *corner detection* algoritme dat gebruikt wordt voor real-time applicaties.

Het gaat als volgt te werk;

1. Overloop alle pixels p, stel we testen een bepaalde pixel px
2. Neem een cirkelvormige omgeving rond de pixel, neem een aantal pixels die op de rand liggen van deze cirkel
3. Pixel px wordt beschouwd als een hoekpunt als n van zijn buren op de cirkel helderder zijn dan $I_{px} + t$ of donkerder dan $I_{px} - t$ (met I_{px} de Intensiteit van pixel px en t een threshold)

Het algoritme werd nog verder geoptimaliseerd om rekening te houden met rotaties en zo rotatie invariant te zijn. Vandaar de naam *oriented* FAST.

2.2.1.2 Rotated BRIEF

Het BRIEF (Binary Robust Independent Elementary Features) algoritme [1] is een descriptor algoritme gebaseerd op het vergelijken van intensiteiten binnen een beschouwde ROI rond een keypoint. Twee punten binnen het beschouwde stukje worden gekozen en de intensiteiten worden vergeleken. Is de intensiteit van het eerste punt groter dan het tweede punt, dan wordt een waarde 1 meegegeven, anders de waarde 0. Zo word een vector bekomen van boolean waardes. Bij het matchen worden de Hamming afstanden vergeleken tussen de vectoren van de boolean waardes. Dit heeft als gevolg en voordeel dat het een snel algoritme is. Het algoritme werd nog verder geoptimaliseerd om rekening te houden met rotaties en zo rotatie invariant te zijn. Vandaar de naam *rotated* BRIEF.

2.2.1.3 Brute Force Matching

De matching van features tussen twee afbeeldingen kan gebeuren door middel van brute force matching. Dankzij ORB kan men de belangrijke punten en descriptoren van deze belangrijke punten op beide afbeeldingen vinden. Aangezien de descriptoren worden bijgehouden als een binaire string, zal bij het vergelijken naar de Hammingafstand gekeken worden. De Hammingafstand is gelijk aan het aantal bits die verschillend zijn in de twee binaire strings.

Men zal op de eerste afbeelding één enkele descriptor van een feature vergelijken met alle andere descriptoren op de andere afbeelding. De beste match zal deze zijn waar de Hamming afstand het kleinst is.

2.2.2 Verkiezingsproces

De bedoeling van de matchingfunctie is om (met een zekere waarschijnlijkheid) te kunnen bepalen voor een foto, in welke zaal deze foto getrokken werd. Een zaal kan gekenmerkt worden door de schilderijen die er hangen. Daarom werden alle schilderijen per zaal in de database geplaatst.

Aan elke vergelijking tussen een schilderij die in de database staat en een input schilderij wordt een score gegeven. Deze score geeft weer hoe de schilderijen op elkaar lijken. In de database staan geen schilderijen maar enkel de descriptoren van deze schilderijen. Met een brute force matcher kunnen de descriptoren van alle schilderijen in de database een voor een worden vergeleken met de descriptoren van het input schilderij. Voor elke vergelijking geeft de brute force matcher een aantal matches terug. Een match bestaat uit een descriptor van het ene schilderij en een descriptor van het ander schilderij. Deze matches zijn niet allemaal even sterk. De score is gelijk aan het aantal matches waarvan de Hammingafstand tussen de twee descriptoren kleiner is dan 32. Het getal 32 werd experimenteel gekozen. Wanneer de scores voor elk schilderij in database bekend zijn, wordt gekeken welke zaal de maximum score bevat.

Bij een 100% juist classificerend programma zou bij elk schilderij uit de input afbeelding dezelfde zaal de hoogste score hebben. Dit is helaas niet het geval. Wanneer schilderijen uit de input afbeelding hun maximum score hebben in verschillende zalen, dan wordt de zaal gekozen met de hoogste score. Want in die zaal staat een schilderij die het meest lijkt op het input schilderij.

2.2.3 Threading

2.2.3.1 In demo bestand

Om te weten welk schilderij het meeste lijkt op een schilderij dat op de testfoto staat, moeten alle schilderijen in de dataset een voor een vergeleken worden. Om dit proces meer parallel te laten verlopen, wordt gebruik gemaakt van threads. Threads zorgen ervoor dat code tegelijk kan draaien door ze te laten lopen op verschillende cores van de processor. In dit python project werd dit

geÃ™plementeerd met de library `âmultiprocessing.poolâ`. Hiermee kan een bepaald aantal threads aangeduid worden die een lijst van taken zullen afhandelen. Een aantal threads of workers aanduiden kan zeer simpel:

$$p = ThreadPool(3)$$

Hier worden drie threads aangeduid om de taken af te handelen. Wanneer op een bepaald frame meerdere paintings herkend worden, wordt het scoringsproces versneld op 2 manieren:

1. 2 workers reserveren die voor een bepaald frame, alle schilderijen in een aparte thread zullen afhandelen (maximum 2 threads tegelijkertijd)
2. 3 workers reserveren die voor een bepaald schilderij, een matchingscore zal berekenen voor alle zalen in een aparte thread (max 3 threads tegelijkertijd)

In python kan een lijst opdrachten meegegeven worden aan workers door een methode template mee te geven en een lijst met inputgegevens voor deze opdracht. Hierna kan het resultaat opgehaald worden als een array van de deelresultaten, in de zelfde volgorde als de inputgegevens. De implementatie in python is vrij simpel:

```
lijstScores = p.map(matchPaintingFunction, paintingDescriptorsList)
```

`paintingDescriptorsList` is een lijst van de descriptoren van de schilderijen die gevonden werden in een bepaalde afbeelding. Dus voor elk schilderij dat gevonden werd zal de `matchPaintingFunction` opgeroepen worden met een item uit `paintingDescriptorsList`. Hierna zullen alle resultaten toegevoegd worden aan `lijstScores`. Uiteindelijk zal `lijstScores` voor elke painting een associatieve array bevatten die voor elke zaal een gelijkheidsscore geeft voor het schilderij.

2.2.3.2 Op testset

Het was nodig ons programma te evalueren en het effect van kleine wijzigingen snel te kunnen waarnemen aan de hand van bepaalde resultaten (HOOFDSTUK 3). Hiervoor moesten een reeks test afbeeldingen, namelijk de test dataset, zo snel mogelijk afgehandeld kunnen worden.

Het doel was hier dus niet meer om zoals in de main file één afbeelding maar om een grote lijst afbeeldingen zo snel mogelijk af te handelen. Hiervoor werd de verdeling van threadpools aangepast. Er werden twee wijzigingen aangebracht:

1. Een schilderij wordt aan de database gematcht in één thread, dus geen aparte thread meer per zaal.
2. 3 workers reserveren die alle afbeeldingen uit de test dataset behandelen en matchen in een aparte thread, maximum 3 threads tegelijkertijd.

2.2.4 Wazige afbeeldingen detecteren

Wanneer men wazige afbeeldingen wilt matchen aan de database komt men vaak verkeerde resultaten uit. Dit is door het feit dat de descriptoren van specifieke

punten in de afbeeldingen die gedetecteerd worden met ORB, andere bitstrings zullen zijn. Bijgevolg zal de Hammingafstand met een gelijkaardige descriptor uit de database groter zijn. Hierdoor zal de matchingscore lager zijn.

In een wazige afbeelding zal het ORB algoritme het ook moeilijker hebben om interessante punten te detecteren, waardoor de matchingscore nog meer zakt. Hierdoor zijn wazige afbeeldingen vrijwel onbruikbaar.

2.2.4.1 Methode

Het doel is om een floating point value terug te krijgen bij een afbeelding die weergeeft hoeveel een afbeelding geblurred is. Dit wordt bereikt met volgende lijn code.

```
cv2.Laplacian(image, cv2.CV_64F).var()
```

Over een afbeelding met grijswaarden wordt de Laplacian kernel (figuur 3) geconvoluteerd. Deze zal per pixel weergeven hoeveel het verschil is in intensiteit met zijn buren. Bij vlakken met dezelfde kleur zal dit dus lage pixel waarden geven.

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Figuur 3: Laplacian kernel

Bij een scherpe afbeelding gaan we er van uit dat de geconvoluteerde afbeelding hoge en lage intensiteiten zal bevatten (scherpe edges en vlakke stukken). Maar wanneer men een wazige afbeelding convoluteert zullen er minder extreme intensiteiten zijn, de intensiteiten liggen dus meestal dichter bij het gemiddelde. Hierdoor zal de variantie (de spreiding van de intensiteiten) lager zijn. Om een wazige afbeelding te detecteren moet de variantie minimum boven een bepaalde waarde zijn.

2.3 Database oprichten

We hebben gekozen om een Database op te richten die alle descriptoren bevat per afbeelding zodat deze niet *at runtime* berekend moeten worden. Om dit te verwezenlijken wordt er gebruik gemaakt van de library 'pickle' om de datastructuur op te slaan als binary file zie de Figuur hier onder.

 database.bin	08/06/2019 14:47	BIN File	11,260 KB
--	------------------	----------	-----------

Als datastructuur werd geopteerd voor een python dictionary met als key een string met de zaal waarin het schilderij zich bevindt en als value een tuple

van de bestandsnaam en de descriptoren van de afbeelding;
 $\{\text{zaal} ; (\text{fileName}, \text{descriptors})\}$

Om de test-database op te stellen is er een script geschreven dat het manueel bewaren van frames mogelijk maakt uit video's die zijn opgenomen in het MSK d.m.v een GoPro camera. Deze frames worden vervolgens in een mappenstructuur geplaatst per zaal waartoe ze behoren, om ze zo gemakkelijk een grondwaarheidslabel te kunnen geven, dit zorgt ervoor dat we daarna resultaten kunnen genereren van hoe nauwkeurig het algoritme is in het matchen van schilderijen uitgesneden uit de test-afbeeldingen met die vanuit de meegegeven dataset van schilderijen.

3 Resultaten

Er zijn twee testsets opgesteld om deze toepassing te evalueren. De eerste testset bestaat uit afbeeldingen met een hogere resolutie die met een smartphone in het museum gemaakt zijn. De tweede testset (video testset) bestaat uit frames die uit 18 video's geknipt zijn die met een smartphone of een GoPro gemaakt zijn. De kwaliteit van de frames ligt een stuk lager dan de kwaliteit van de afbeeldingen, dit zal dan ook verklaren waarom de toepassing beter presteert op de eerste testset.

De metrieken die gebruikt worden om de toepassing kwantitatief te evalueren zijn: precision, recall en accuracy en worden berekend volgens formules waarbij TP, FP, TN en FN respectievelijk de true positives, false positives, true negatives en false negatives zijn:

$$\text{precision} = \text{TP}/(\text{TP} + \text{FP}) \quad (1)$$

$$\text{recall} = \text{TP}/(\text{TP} + \text{FN}) \quad (2)$$

$$\text{accuracy} = (\text{TP} + \text{TN})/(\text{TP} + \text{TN} + \text{FP} + \text{FN}) \quad (3)$$

Eerst wordt een dictionary gemaakt met als key een bestandsnaam en als value de zaal waar dit bestand bij hoort. De mogelijke schilderijen worden uitgeknipt en gematcht met de databank, deze match wordt dan vergeleken met de ground truth. De video testset bevat 614 afbeeldingen en de andere testset met foto's van hogere kwaliteit bevat 80 afbeeldingen. De resultaten staan in tabel 1.

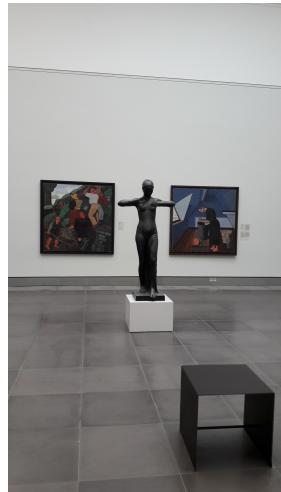
Als de er geen mogelijke schilderijen worden gevonden of als de afbeelding te onscherp is, wordt deze niet verwerkt en ook niet meegerekend in de metrieken. De toepassing presteert aanzienlijk beter op de testset met foto's van hogere kwaliteit, dit was te verwachten omdat hier minder nuttige keypoints op gevonden worden. In de testset met 80 afbeeldingen werden 0 afbeeldingen niet verwerkt omdat deze te wazig waren en op 14 afbeeldingen werden er geen mogelijke schilderijen gevonden. Dit betekent dat op 82% van de afbeeldingen mogelijke schilderijen werden gevonden. In de testset met 614 afbeeldingen werden

	Testset Minerva	Testset video's
Precision	0.89	0.72
Recall	1.00	0.97
Accuracy	0.89	0.70
Too blurry	0 afbeeldingen	95 afbeeldingen
Didn't find any paintings	14 afbeeldingen	58 afbeeldingen
Size	80 afbeeldingen	614 afbeeldingen

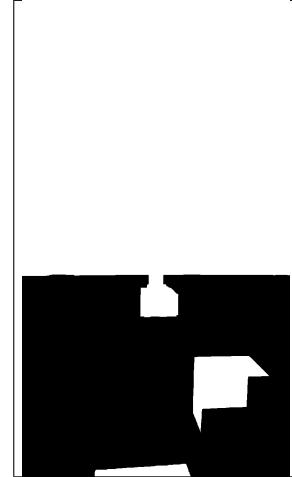
Tabel 1: Resultaten testsets

95 afbeeldingen niet verwerkt omdat deze te wazig waren en op 58 afbeeldingen werden er geen mogelijke schilderijen gevonden. Dit betekent dat op 89% van de afbeeldingen mogelijke schilderijen werden gevonden.

De methode om schilderijen te segmenteren werkt enkel optimaal onder bepaalde voorwaarden. De achtergrond moet het grootste oppervlak zijn in de afbeelding. Figuur 4 (a) is een voorbeeld waar het segmenteren niet correct gebeurt omdat de vloer een groot deel van de afbeelding inneemt. De mask die gegenereerd wordt staat in figuur 4 (b)



(a) originele afbeelding.



(b) gegenereerde mask.

Figuur 4: vergelijking origineel en slecht gegenereerde mask

4 Conclusie

Als uiteindelijke accuracy behaalt het algoritme 89% op de testset van Minerva en 70% op de testset gegenereerd uit de GoPro video's. Het algoritme zal wel niet werken als het schilderij een omlijsting heeft die niet vierhoekig is of als de omlijstingen van verschillende schilderijen elkaar overlappen. Het algoritme werkt het beste wanneer een schilderij aan een effen muur hangt. Als een schilderij te dicht bij de rand van de afbeelding voorkomt of als de muur niet het grootste segment is, is het algoritme minder performant.

Bibliografie

- [1] Calonder, M., Lepetit, V., Strecha, C., Fua, P.: Brief: Binary robust independent elementary features. In: Computer Vision - ECCV 2010. vol. 6314, pp. 778 – 792 (09 2010). https://doi.org/10.1007/978-3-642-15561-1_56
- [2] Rublee, E., Rabaud, V., Bradski, G.: Orb: An efficient alternative to sift or surf. In: 2011 International Conference on Computer Vision. pp. 2564 – 2571 (11 2011). <https://doi.org/10.1109/ICCV.2011.6126544>
- [3] Natin, G.: Locating recognising paintings in galleries (December 2017), <https://github.com/nating/recognizing-paintings>
- [4] Szeliski, R.: Computer Vision: Algorithms and Applications. Springer (09 2010). <https://doi.org/10.1007/978-1-84882-935-0>
- [5] Tyagi, D.: Phage lambda: description & restriction map (January 2019), <https://medium.com/software-incubator/introduction-to-orb-oriented-fast-and-rotated-brief-4220e8ec40cf>